
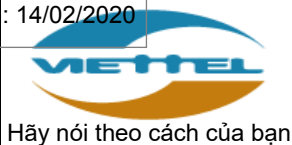


STT	Người ký	Đơn vị	Thời gian ký	Ý kiến
1	TỔNG CÔNG TY GIẢI PHÁP DOANH NGHIỆP VIETTEL - CHI NHÁNH TẬP ĐOÀN CÔNG NGHIỆP - VIỆN THÔNG QUẢN ĐỘI		14/02/2020 16:30:41	Đã đóng dấu 
2	LÊ THÀNH CÔNG	Phó Tổng Giám đốc - Ban Tổng Giám đốc - Tổng công ty Giải pháp doanh nghiệp Viettel	14/02/2020 15:22:22	
3	TRƯƠNG THANH HẢI	Phó Giám đốc Trung tâm - Trung tâm Giải pháp - Tổng công ty Giải pháp doanh nghiệp Viettel	14/02/2020 12:34:12	

Mã văn bản: HD.VTS.GP.12

Số văn bản: 12

Ngày ban hành: 14/02/2020



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

Mã hiệu: HD.VTS.GP.

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 1/49

BẢNG THEO DÕI SỬA ĐỔI

STT	Trang	Nội dung sửa đổi	Ngày có hiệu lực
1	Toàn bộ	Ban hành tài liệu lần 1	01/03/2020

	Biên soạn	Kiểm tra	Phê duyệt
Chữ ký		TRUNG TÂM GIẢI PHÁP  Trương Thanh Hải	PHÓ TỔNG GIÁM ĐỐC  Thiếu tá Lê Thành Công

**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL****HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 2/49

I. Mục đích:

Tài liệu này được xây dựng nhằm hướng dẫn người dùng hiểu và thực hiện triển khai áp dụng tích hợp liên tục/triển khai liên tục (CI/CD) trong quy trình phát triển phần mềm nhằm nâng cao hiệu quả của quá trình sản xuất phần mềm, nâng cao chất lượng sản phẩm (SP) cũng như rút ngắn thời gian phát hành nhằm đáp ứng một cách nhanh nhất các yêu cầu của kinh doanh, nghiệp vụ.

II. Phạm vi áp dụng:

Tài liệu này phục vụ cho các thành viên đội dự án tham gia phát triển và nâng cấp phần mềm thuộc Tổng Công Ty.

III. Định nghĩa và thuật ngữ: N/A**IV. Nội dung:****1. Giới thiệu chung****1.1. Tích hợp liên tục/Triển khai liên tục (CI/CD)**

Continuous Integrations (CI) - Tích hợp liên tục, là một phương pháp phát triển phần mềm (PM) mà những người phát triển trong dự án PM sẽ tích hợp sự thay đổi lên nhánh chính (trong git là master) trên kho mã nguồn của dự án một cách liên tục và thường xuyên, với tần suất được khuyến nghị là ít nhất 1 lần/ngày. Trong môi trường sử dụng Git làm repository, nhà phát triển thực hiện tích hợp bằng cách tạo ra các commit, và merge request để tích hợp sự thay đổi vào nhánh chính.

Triển khai liên tục (Continuous Delivery – CD) là nguyên tắc phát triển PM trong đó PM được có thể phát hành và triển khai lên môi trường production bất kỳ lúc nào trong suốt quá trình phát triển (Martin Fowler). Continuous Delivery là sự kéo dài của CI theo hướng đội dự án đảm bảo mỗi thay đổi (commit) đối với hệ thống đều có thể phát hành được (releaseable) và chỉ cần nhấn một nút để thực hiện toàn bộ quá trình phát hành. Từ đó CI/CD tạo thành một luồng khép kín từ giai đoạn phát triển đến khi đưa SP đến người dùng cuối.

Mỗi sự tích hợp do những người phát triển tạo ra đều được kiểm thử và xác nhận tự động bởi một hệ thống CI/CD. Hệ thống CI/CD sẽ tự động phát hiện sự thay đổi về mã nguồn khi nhà phát triển đẩy các commit lên các nhánh của repository, sau đó tùy thuộc vào sự thay đổi mà người phát triển tạo ra liên quan đến việc tích hợp vào nhánh phát triển (push commit) hay tích hợp vào nhánh chính (merge request) mà hệ thống CI/CD sẽ lựa chọn hình thức kiểm thử tích



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 3/49

hợp phù hợp. Nếu hệ thống CI/CD phát hiện lỗi xảy ra với mã nguồn trong quá trình kiểm thử tự động, hệ thống sẽ gửi thông báo tới người phát triển.

1.2. Các thay đổi mà CI đem tới cho quá trình phát triển phần mềm

Áp dụng phương thức phát triển tích hợp liên tục, nhóm phát triển chia quá trình phát triển tính năng/fix bug/hotfix thành các giai đoạn nhỏ, mỗi một giai đoạn nhỏ bao gồm việc phát triển một phần của tính năng rồi tích hợp vào repository, thay vì việc phát triển toàn bộ tính năng trong 1 thời gian dài, sau đó tích hợp một khối lượng lớn mã nguồn đã phát triển vào repository.

Việc mỗi chu kỳ phát triển - tích hợp có thời gian ngắn đồng nghĩa với việc khối lượng mã nguồn hệ thống đã thay đổi có kích thước nhỏ, khối lượng mã nguồn thay đổi trong 1 lần tích hợp nhỏ cho phép các nhà phát triển dễ dàng kiểm soát những thay đổi gì đã được thực hiện lên mã nguồn, dễ dàng viết các bài test tự động cho các thay đổi mã nguồn nhỏ, dễ dàng phát hiện các lỗi của mã nguồn. Lý do, đó là việc quản lý, kiểm soát và viết mã test tự động cho một khối lượng mã nguồn thay đổi nhỏ với kích thước khoảng vài trăm dòng luôn dễ dàng hơn rất nhiều so với việc phải rà soát và kiểm tra một khối mã nguồn đã bị thay đổi vài nghìn dòng.

Việc thường xuyên tích hợp cho phép mã nguồn được thường xuyên kiểm thử tự động khi xảy ra các thay đổi. Kiểm thử tự động cho phép phát hiện các lỗi xảy ra khi mã nguồn bị thay đổi thông qua việc thực hiện các bài test, từ đó hệ thống này sẽ thông báo cho nhà phát triển về các lỗi đã xuất hiện. Và với kích thước mã nguồn bị thay đổi là nhỏ, nhà phát triển có thể dễ dàng xử lý các lỗi đó.

Mã nguồn được tích hợp liên tục mỗi ngày cũng cho phép các nhà phát triển trong nhóm liên tục thấy được các thay đổi mà các người phát triển khác trong nhóm đã tạo ra trên mã nguồn chung, từ đó có thể phát hiện các thay đổi, xung đột giữa mã nguồn của họ với các nhà phát triển khác trên hệ thống. Các thay đổi là nhỏ, do vậy việc xử lý xung đột sẽ đơn giản, dễ dàng và nhanh chóng hơn rất nhiều so với việc 2 lập trình viên ngồi phát triển hai khối mã nguồn hàng nghìn dòng trên hai nhánh độc lập, sau đó tích hợp vào nhánh chính ở những ngày cuối của milestone và tìm cách xử lý hàng loạt các xung đột xảy ra khi tích hợp hai khối code khổng lồ với nhau vào nhánh chính.

Để các nhà phát triển có thể tích hợp các thay đổi lên nhánh mã nguồn chính liên tục với chu kỳ ngắn, đòi hỏi việc phân tích yêu cầu và chức năng phải phân chia công việc trong quá trình phát triển thành các phần đủ nhỏ, để cho phép



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 4/49

những người phát triển có thể hoàn thành phần công việc được giao trong một khoảng thời gian ngắn, tối đa là từ 1 đến hai ngày, sau đó tích hợp ngay phần công việc đã hoàn thành lên nhánh mã nguồn chính thông qua các công cụ như pull request và merge request.

1.3. Các điều kiện cần thiết để triển khai CI/CD

Duy trì một nhánh chính (master) và một nhánh phát triển (dev) trên repository. Các thành viên nên thực hiện việc tích hợp kết quả công việc từ nhánh phát triển tính năng (feature) vào nhánh dev (merge nhánh tính năng vào nhánh phát triển) ít nhất 1 lần 1 ngày.

Các nhà phát triển cần tích hợp code vào mã nguồn thường xuyên và liên tục với kích thước vừa phải: Mã nguồn có kích thước vừa phải sẽ giúp cho tất cả mọi người có thể thường xuyên theo dõi và kiểm soát được tất cả các thay đổi đã xảy ra trên mã nguồn.

Phát triển mã nguồn và mã kiểm thử của mã nguồn vừa viết trong một chu kỳ tích hợp: Khi áp dụng tích hợp liên tục, việc kiểm thử đơn thuần bằng con người là không còn phù hợp, do số lượng tích hợp xảy ra trong một ngày là rất lớn, vượt quá khả năng xử lý của con người. Vì vậy ở quá trình tích hợp phải áp dụng kiểm thử tự động thay cho kiểm thử bằng con người, điều này đồng nghĩa với việc nhà phát triển sẽ thực hiện đồng thời việc phát triển mã nguồn với việc viết mã kiểm thử cho mã nguồn ấy trong một chu kỳ tích hợp, sao cho mã kiểm thử có thể xác nhận chất lượng của mã nguồn đã phát triển đáp ứng các yêu cầu đặt ra.

Đảm bảo quá trình kiểm thử diễn ra nhanh, trả về kết quả kiểm thử trong thời gian ngắn, để nhà phát triển có thể phát hiện ra các vấn đề xảy ra một cách sớm nhất có thể. Để tối ưu hóa quá trình kiểm thử, các bài kiểm thử nhanh, tốn ít thời gian (unit test) sẽ được thực hiện trước, các bài kiểm thử phức tạp, tốn nhiều thời gian (integration test, end to end test) sẽ được thực hiện sau.

Kiểm soát và xử lý các lỗi xảy ra trên mã nguồn ngay khi nhận được thông báo phát hiện ra lỗi từ quá trình kiểm thử tự động. Kiểm thử trước khi tích hợp thay đổi từ nhánh phát triển tính năng (feature) vào nhánh phát triển (dev), sao cho hạn chế tối đa việc xảy ra các lỗi trên mã nguồn nhánh chính khi kiểm thử. Việc xảy ra lỗi trên mã nguồn nhánh chính đồng nghĩa với việc tất cả mọi người trong nhóm phát triển phải dừng việc tích hợp lại và chờ đợi mã nguồn nhánh chính được fix hết lỗi, sau đó mới có thể tiếp tục quá trình tích hợp. Để làm



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

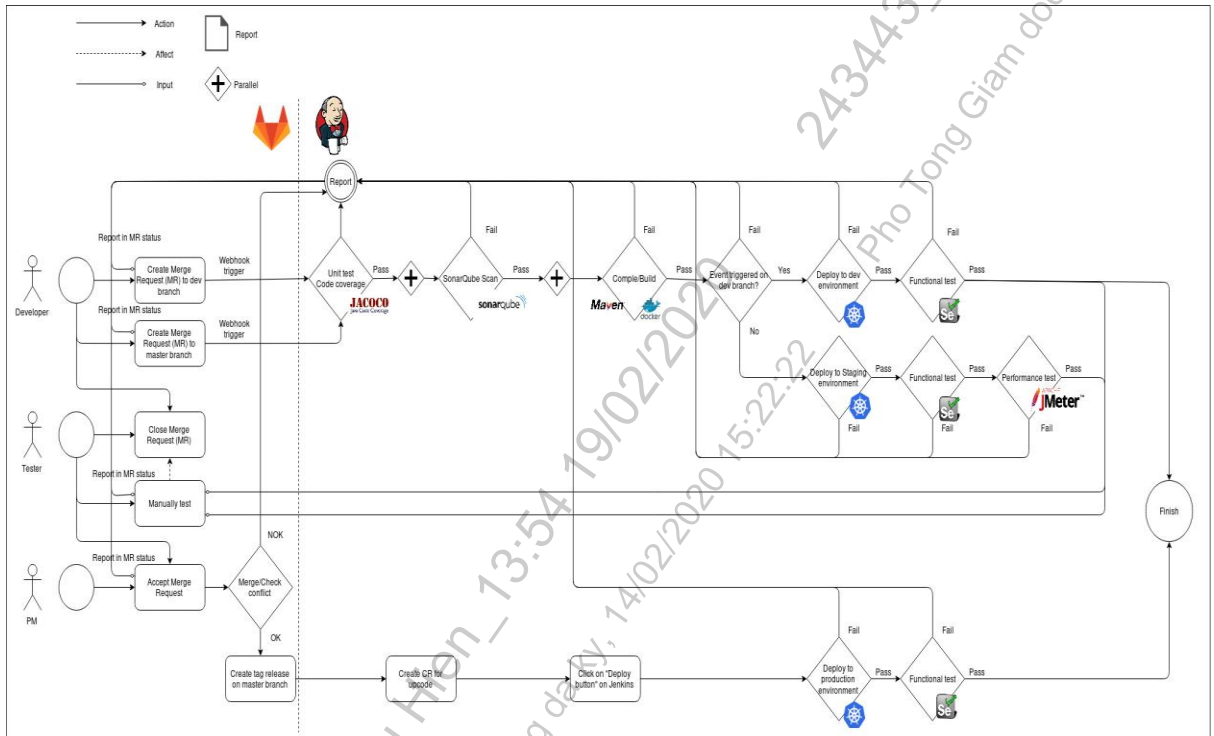
Lần ban hành: 01

Trang/tổng số trang: 5/49

được điều này, nhà phát triển cần đặt ưu tiên cho việc fix các lỗi xảy ra hơn tất cả các việc khác.

1.4. Luồng CI/CD tiêu chuẩn

Dưới đây là luồng CI/CD chuẩn đang được áp dụng cho hệ thống thinghub.



Các thành phần của hệ thống CI/CD cho dự án thinghub:

Để áp dụng mô hình phát triển phần mềm CI/CD cho dự án thinghub, một nền tảng phát triển bao gồm các thành phần dưới đây đã được xây dựng:

- Hệ thống quản lý dự án, lưu trữ mã nguồn và cấu hình hệ thống: Gitlab.
- Hệ thống CI/CD thực hiện việc build/test tự động và triển khai mã nguồn lên môi trường test, staging và môi trường production: Jenkins
- Hệ thống lưu trữ các sản phẩm phần mềm (artifact) và các thư viện được sử dụng trong hệ thống Tiêm chủng và các dự án khác: Nexus Repository và Habor cho hệ thống thinghub
- Hệ thống thông báo kết quả build: Viettel Mail.

1.5. Luồng CI của hệ thống Thinghub

1.5.1. Các luồng xử lý phục vụ cho quá trình tích hợp liên tục – CI

Với dự án **Thinghub**, để thực hiện công việc tích hợp liên tục, hiện tại hệ thống CI/CD có hai luồng xử lý chính tương ứng với hai loại sự kiện sau:

- Sự kiện mã nguồn trong **nhánh phát triển tính năng (feature branch)** thay đổi khi nhà phát triển đã bắt đầu quá trình tích hợp mã nguồn - nhà phát triển đã tạo merge request từ **nhánh phát triển tính năng (feature branch)** tới



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

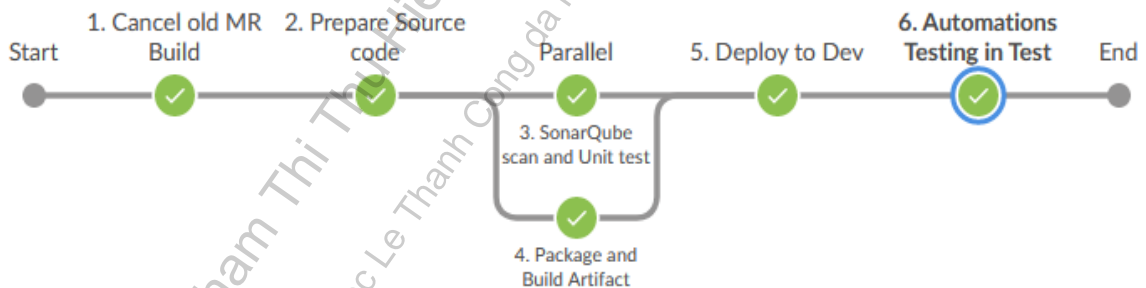
Lần ban hành: 01

Trang/tổng số trang: 6/49

nhánh phát triển chung (develop branch). Lúc này luồng xử lý *merge request build* được sử dụng. Trong trường hợp nhánh đích của merge request thay đổi, luồng xử lý merge request build cũng được kích hoạt, yêu cầu cần kiểm tra lại kết quả sau tích hợp giữa nhánh phát triển tính năng và nhánh phát triển chung có vấn đề gì không. → Khi có sự kiện merge request build từ nhánh phát triển tính năng (feature branch) tới nhánh phát triển chung (develop branch) luồng **CI dev** sẽ được thực hiện.

- Sự kiện mã nguồn trong nhánh phát triển chung (develop branch) thay đổi khi sự kiện merge request build của **CI dev** được accept. Nhà phát triển bắt đầu quá trình tích hợp mã nguồn tức là đã tạo merge request từ nhánh phát triển chung (develop branch) tới nhánh chính (master branch). Lúc này luồng xử lý *merge request build* được sử dụng. Trong trường hợp nhánh đích của merge request thay đổi, luồng xử lý *merge request build* cũng được kích hoạt, yêu cầu cần kiểm tra lại kết quả sau tích hợp giữa nhánh phát triển chung và nhánh phát triển chính có vấn đề gì không. → Khi có sự kiện *merge request build* từ nhánh phát triển chung (develop branch) tới nhánh chính (master branch) luồng **CI staging** sẽ được thực hiện.

a. Luồng Merge Request Build – CI-dev



Luồng xử lý merge request build diễn ra các hoạt động sau:

- Sự kiện kích hoạt luồng: **Nhà phát triển** có thể kích hoạt luồng merge request build bằng một trong các cách sau:
 - Nhà phát triển tạo một merge request mới.
 - Mã nguồn của nhánh nguồn hoặc nhánh đích của một quá trình tích hợp - merge request bị thay đổi.
 - Người dùng sử dụng chuỗi ký tự đặc biệt (hiện tại là **recheck**) comment vào Merge Request để yêu cầu build lại Merge Request.
- Hệ thống quản lý mã nguồn Gitlab gửi thông điệp trigger hệ thống CI/CD, với tham số đầu vào là tên nhánh đích, tên nhánh nguồn của merge request và các HEAD commitID trên các nhánh này tại thời điểm kích hoạt trigger.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 7/49

3. Dựa vào các tham số đầu vào được **Gitlab** gửi qua webhook và thông tin cấu hình trên các job ở Jenkins, **Jenkins Master** lựa chọn ra job, node phù hợp.

4. **Jenkins Master** chuyển tiếp build tới **Build Machine Jenkins Slaves** để tiến hành các bước kiểm thử.

5. Hệ thống CI/CD ngừng tất cả các merge request build đang build cho merge request này, sau đó thực hiện lấy mã nguồn của nhánh nguồn và mã nguồn nhánh đích từ **Gitlab** rồi tích hợp chúng lại với nhau để tạo ra mã nguồn sau tích hợp.

6. Tại **Build Machine Jenkins Slaves**, hệ thống CI/CD tiến hành thực hiện các bước để kiểm thử mã nguồn, bao gồm: biên dịch mã nguồn, test đơn vị (unit test), test độ bao phủ kiểm thử (code coverage test), test an toàn thông tin cho mã nguồn (sonarqube scan).

7. Nếu pass qua các bài test kiểm thử trên, hệ thống CI/CD thực hiện deploy tới môi trường test và thực hiện test tự động tính năng (Automations Testing) trên môi trường tests.

8. **Build Machine Jenkins Slaves** gửi kết quả test về cho **Jenkins Master**

9. Hệ thống CI/CD trả kết quả merge request commit build về cho hệ thống quản lý dự án **Gitlab**, thông báo kết quả build merge request, bao gồm các nội dung sau:

Kết quả kiểm thử cho mã nguồn trên nhánh phát triển ở đúng commitID đã được truyền lên. Kết quả kiểm thử bao gồm các thông tin chi tiết về kết quả các bài test, và kết quả tổng thể của push commit Build, bao gồm:

- Số lượng unit test thành công/thất bại
- Độ bao phủ kiểm thử của mã nguồn trên nhánh phát triển (code coverage) theo tỉ lệ %
- Số lượng các lỗi an toàn thông tin quét SonarQube
- Kết quả Automations Test trên jenkins
- Địa chỉ của môi trường tests.
- Kết quả tổng thể của Merge request build: Nếu tất cả các bước 2,3,4,5,6 đều không có vấn đề, merge request build có kết quả là thành công- success. Nếu không, merge request build có kết quả là thất bại.

10. Hệ thống **Gitlab** gửi thông báo về kết quả merge request build tới nhà phát triển sử dụng **Viettel Mail Sever**.

11. **Nhà phát triển** check mail để nhận thông báo kết quả.

b. Luồng Merge Request Build – CI-staging



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

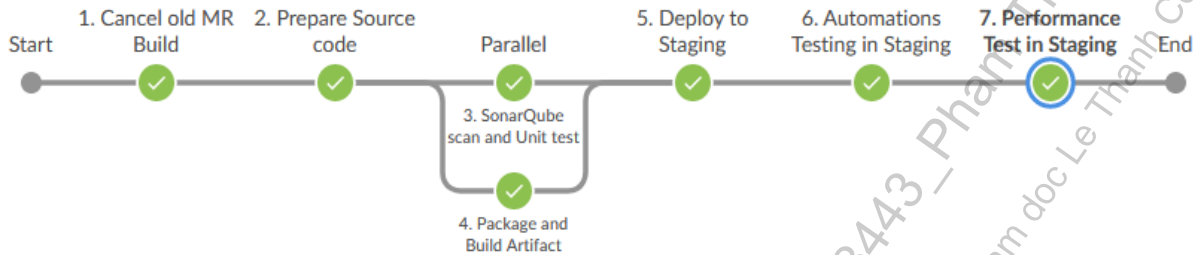
Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 8/49



Luồng xử lý merge request build diễn ra các hoạt động sau:

1. Sự kiện kích hoạt luồng: **Nhà phát triển** có thể kích hoạt luồng merge request build bằng một trong các cách sau:
 - o Nhà phát triển tạo một merge request mới.
 - o Mã nguồn của nhánh nguồn hoặc nhánh đích của một quá trình tích hợp - merge request bị thay đổi.
 - o Người dùng sử dụng chuỗi ký tự đặc biệt (hiện tại là **recheck**) comment vào Merge Request để yêu cầu build lại Merge Request.
2. Hệ thống quản lý mã nguồn Gitlab gửi thông điệp trigger hệ thống CI/CD, với tham số đầu vào là tên nhánh đích, tên nhánh nguồn của merge request và các HEAD commitID trên các nhánh này tại thời điểm kích hoạt trigger.
3. Dựa vào các tham số đầu vào được **Gitlab** gửi qua webhook và thông tin cấu hình trên các job ở Jenkins, **Jenkins Master** lựa chọn ra job, node phù hợp.
4. **Jenkins Master** chuyển tiếp build tới **Build Machine Jenkins Slaves** để tiến hành các bước kiểm thử.
5. Hệ thống CI/CD ngừng tất cả các merge request build đang build cho merge request này, sau đó thực hiện lấy mã nguồn của nhánh nguồn và mã nguồn nhánh đích từ **Gitlab** rồi tích hợp chúng lại với nhau để tạo ra mã nguồn sau tích hợp.
6. Tại **Build Machine Jenkins Slaves**, hệ thống CI/CD tiến hành thực hiện các bước để kiểm thử mã nguồn, bao gồm: biên dịch mã nguồn, test đơn vị (unit test), test độ bao phủ kiểm thử (code coverage test), test an toàn thông tin cho mã nguồn (sonarqube scan).
7. Nếu pass qua các bài test kiểm thử trên, hệ thống CI/CD thực hiện deploy tới môi trường staging và thực hiện test tự động tính năng (Automations Testing) trên môi trường staging.
8. Nếu kết quả test tự động tính năng pass thì sẽ tiến hành test tự động hiệu năng trên môi trường staging (Auto Performance).



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 9/49

9. **Build Machine Jenkins Slaves** gửi kết quả test về cho **Jenkins Master**

10. Hệ thống CI/CD trả kết quả merge request commit build về cho hệ thống quản lý dự án **Gitlab**, thông báo kết quả build merge request, bao gồm các nội dung sau:

Kết quả kiểm thử cho mã nguồn trên nhánh phát triển ở đúng commitID đã được truyền lên. Kết quả kiểm thử bao gồm các thông tin chi tiết về kết quả các bài test, và kết quả tổng thể của push commit Build, bao gồm:

- Số lượng unit test thành công/thất bại
- Độ bao phủ kiểm thử của mã nguồn trên nhánh phát triển (code coverage) theo tỉ lệ %
- Số lượng các lỗi an toàn thông tin quét SonarQube.
- Kết quả Automations Test (Functions Test và Performance Test) trên Jenkins.
- Địa chỉ của môi trường staging.
- Kết quả tổng thể của Merge request build: Nếu tất cả các bước 2,3,4,5,6 đều không có vấn đề, merge request build có kết quả là thành công- success. Nếu không, merge request build có kết quả là thất bại.

11. Hệ thống **Gitlab** gửi thông báo về kết quả merge request build tới nhà phát triển thông qua hệ thống **Viettel Mail Sever**.

12. **Nhà phát triển** check mail để nhận thông báo kết quả.

1.5.2. Kịch bản nhà phát triển thực hiện phát triển phần mềm triển khai theo mô hình CI

Sử dụng các luồng xử lý của nền tảng phát triển, nhà phát triển thực hiện phát triển phần mềm theo phương pháp tích hợp liên tục trong dự án theo kịch bản sau.

a. Quá trình phát triển tính năng trên nhánh phát triển

Sau khi tính năng đã được hoàn thiện trên nhánh phát triển tính năng. Nhà phát triển bắt đầu quá trình tích hợp mã nguồn nhánh phát triển tính năng vào mã nguồn nhánh phát triển chung để đưa tính năng mới lên server để kiểm thử. Quá trình tích hợp bắt đầu khi nhà phát triển tạo yêu cầu tích hợp trên Gitlab bằng cách tạo ra một Merge Request từ nhánh phát triển tính năng (feature branch) tới nhánh phát triển chung (dev branch). Khi merge request được tạo ra, luồng xử lý merge request build sẽ được thực hiện. Nếu merge request build phát hiện lỗi xảy ra trong quá trình tích hợp, như:

- Xung đột - conflict giữa mã nguồn nhánh phát triển tính năng và nhánh phát triển chung.
- Kết quả kiểm thử thất bại.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 10/49

thì nhà phát triển cần xử lý các lỗi này và cập nhật lại mã nguồn nhánh phát triển tính năng. Cho đến khi luồng xử lý merge request build thành công trên môi trường test. Lúc này người kiểm thử - tester hoặc người phát triển tính năng đó sẽ tham gia vào kiểm thử để xác nhận tính năng mới đã đạt các yêu cầu đặt ra. Đồng thời, trưởng nhóm phát triển (PM) sẽ tham gia vào review code. Nhà phát triển tiếp tục cập nhật mã nguồn nhánh phát triển tính năng, cho đến khi người trưởng nhóm và tester đồng ý tính năng mới đã đạt yêu cầu. Lúc này trưởng nhóm sẽ thực hiện chấp nhận merge request, mã nguồn nhánh phát triển tính năng được tích hợp vào nhánh phát triển chung và quá trình tích hợp kết thúc.

b. Quá trình tích hợp tính năng vào nhánh chính của mã nguồn

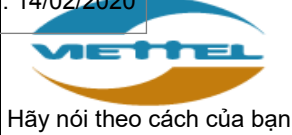
Sau khi tính năng đã được hoàn thiện và merge vào nhánh phát triển chung. Nhà phát triển bắt đầu quá trình tích hợp mã nguồn nhánh phát triển chung vào mã nguồn nhánh chính để đưa tính năng mới ra phục vụ người dùng. Quá trình tích hợp bắt đầu khi nhà phát triển tạo yêu cầu tích hợp trên Gitlab bằng cách tạo ra một Merge Request từ nhánh phát triển chung (dev branch) tới nhánh chính (master branch). Khi merge request được tạo ra, luồng xử lý merge request build sẽ được thực hiện. Nếu merge request build phát hiện lỗi xảy ra trong quá trình tích hợp, như:

- Xung đột - conflict giữa mã nguồn nhánh phát triển và nhánh chính.
- Kết quả kiểm thử thất bại.

thì nhà phát triển cần xử lý các lỗi này và cập nhật lại mã nguồn nhánh phát triển. Cho đến khi luồng xử lý merge request build thành công trên môi trường staging. Lúc này người kiểm thử - tester sẽ tham gia vào kiểm thử để xác nhận tính năng mới đã đạt các yêu cầu đặt ra. Đồng thời, trưởng nhóm phát triển (PM) sẽ tham gia vào review code. Nhà phát triển tiếp tục cập nhật mã nguồn nhánh phát triển, cho đến khi người trưởng nhóm và tester đồng ý tính năng mới đã đạt yêu cầu. Lúc này trưởng nhóm sẽ thực hiện chấp nhận merge request, mã nguồn nhánh phát triển chung được tích hợp vào nhánh chính và quá trình tích hợp kết thúc.

1.6. Luồng CD của hệ thống Thinghub

Sau khi nhánh chính mã nguồn của dự án đã được tích hợp thêm các tính năng mới, tại các thời điểm thích hợp quản lý dự án sẽ thực hiện việc build mã nguồn của dự án thành chương trình và triển khai ra môi trường production để phục vụ cho người dùng cuối.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

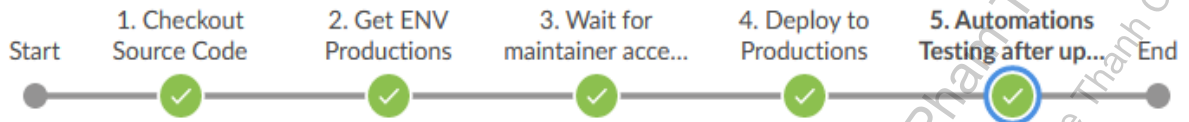
Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 11/49



Các bước tiến hành release như sau:

- Checkout nhánh mới từ master và đặt tên VD: 1.0, hay 1.1
- Tiến hành tạo **tag** tương ứng với version release (VD: tag với tên v1.0) và điền các thông tin liên quan đến bản release như message và changelog
- Chờ Jenkins tự trigger release

2. Phạm vi áp dụng CI/CD

Quan điểm về phạm vi áp dụng CI/CD trong quá trình phát triển phần mềm tại Tổng công ty như sau:

- Trung tâm Sản phẩm (TTSP): áp dụng triển khai CI/CD cho toàn bộ các sản phẩm đại trà.
- Trung tâm Công nghệ thông tin (TT CNTT), Trung tâm Giải pháp (TT GP); áp dụng triển khai CI/CD với toàn bộ các sản phẩm/dịch vụ mới.

Với các sản phẩm/dự án mới của TTGP:

- Tất cả các sản phẩm/dự án mới đều mặc định áp dụng CI/CD, đưa chi phí vào phương án kinh doanh.
- Nếu sản phẩm/dự án mới không áp dụng CI/CD thì bắt buộc phải ghi rõ lý do, đánh giá ảnh hưởng đến chất lượng sản phẩm trong phương án kinh doanh để báo cáo xin ý kiến Ban Tổng Giám đốc.

3. Hướng dẫn ước lượng nỗ lực cho các dự án PM khi triển khai CI/CD

Trong năm 2020, với các dự án triển CI/CD level 2 trở lên yêu cầu thực hiện unit test >50% các chức năng phát triển mới, nỗ lực unit test được tính tối đa bằng 30% nỗ lực phát triển của chức năng thực hiện unit test.

4. Hướng dẫn triển khai CI/CD cho một dự án cụ thể

4.1. Hướng dẫn CI Pipeline cho một dự án cụ thể

Trong phần này hướng dẫn thiết lập các luồng xử lý tích hợp liên tục - CI pipelines cho một dự án phần mềm.

Xuyên suốt tài liệu hướng dẫn, một dự án Java sử dụng **Maven multi-module** để quản lý - dự án thinghub, sẽ được sử dụng để làm ví dụ minh họa cho nội dung này.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 12/49

Các dự án có ngôn ngữ lập trình khác, có project layout khác với dự án được lấy ra làm ví dụ vẫn có thể áp dụng các thiết kế và giải pháp CI/CD đã áp dụng cho hệ thống thinghub, nhưng sẽ cần chỉnh sửa thiết kế bên trong từng bước (stage) cho phù hợp với ngôn ngữ, công nghệ và project layout mà dự án sử dụng.

Một luồng xử lý CI/CD bao gồm các hoạt động sau:

- Người dùng tạo trên hệ thống GitLab một sự kiện CI/CD thông qua một trong số các hành động sau:

- Push commits lên branch.
- Tạo ra merge request. Đóng merge request, cập nhật merge request.
- Tạo Project Tag.
- Comment trong merge request.
- ...

- Khi một sự kiện CI/CD xuất hiện, GitLab tạo ra một event và gửi event đó tới Jenkins thông qua HTTP REQUEST.

- Dựa vào loại event và nội dung của event được GitLab gửi lên, Jenkins GitLab plugin tìm ra Jenkins Job chịu trách nhiệm xử lý event này và kích hoạt một build mới của Jenkins Job này để xử lý event.

- Jenkins Build được tạo ra thực thi script xử lý được định nghĩa trong cấu hình của Jenkins Job.

- Dựa vào kết quả thực thi script, Jenkins Build cập nhật kết quả build lên GitLab thông qua các công cụ như commit status, Merge Request comments, issue.

- GitLab sử dụng hệ thống email của tập đoàn để gửi email thông báo kết quả xử lý sự kiện CI/CD của Jenkins Build cho những người có liên quan tới sự kiện CI/CD được tạo ra trên GitLab.

4.1.1. Các sự kiện và các luồng xử lý CI/CD chính của một Project

Trong quá trình phát triển và vận hành một dự án phần mềm, các sự kiện CI/CD sau được người dùng tạo ra:

- Sự kiện người dùng cập nhật một branch trên GitLab Repo bằng cách đẩy các commit mới lên branch này: **Push commit event**.

- Sự kiện người dùng tạo merge request mới và sự kiện source hoặc target Branch của một Open Merge Request event được thay đổi: **Open Merge Request event**.

- Sự kiện người dùng Accept/Merge một merge Request: **Accept Merge Request event**.

- Sự kiện người dùng Close một merge Request: **Close Merge Request event**


**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 13/49

- Sự kiện người dùng Comment vào một Open Merge Request: **Note Event**
- Sự kiện người dùng tạo ra một Tag mới cho Project: **Tag Push Event**.

Các luồng xử lý được tạo ra để xử lý các event trên:

- Luồng xử lý **Push Commit Build**: Xử lý Sự kiện **Push commit event**
- Luồng xử lý **Open Merge Request Build**: Xử lý Sự kiện **Open Merge Request event** và **Merge Request Note event (Rebuild Merge Request event)**
- Luồng xử lý **Accept/Close Merge Request Build**: Xử lý các Sự kiện **Accept Merge Request event** và **Close Merge Request event**.
- Luồng xử lý **Deploy to Production**: Xử lý **Tag Push Event**

Trong 4 luồng xử lý trên, 3 luồng xử lý đầu thuộc về phần **CI**, còn luồng xử lý **Deploy to Production** thuộc phần **CD**. Trong phần này hướng dẫn thiết lập các luồng xử lý tích hợp liên tục - CI pipelines, cho một dự án phần mềm, do đó trong phần này sẽ hướng dẫn cách xây dựng và thiết lập luồng xử lý trong luồng tích hợp liên tục, đó là các luồng **Open Merge Request Build**. Khi thực hiện merge Request Build vào hai nhánh tích hợp trên gitlab là dev branch và master branch sẽ có hai CI pipeline tương ứng chạy trên hai môi trường khác nhau là môi trường test (**CI_dev**) và môi trường staging (**CI_staging**).

4.1.2. Thiết lập các cấu hình ở các thành phần trong hệ thống CI/CD

4.1.2.1. Cấu hình jenkins

Chịu trách nhiệm chính trong việc xử lý các luồng xử lý CI/CD trong hệ thống là các Jenkins Job. Một Jenkins Job định nghĩa ra các điều kiện, hay các sự kiện sẽ kích hoạt một luồng xử lý của Jenkins - được gọi là Jenkins Build, và script xử lý mà Jenkins Build đó sẽ thực thi sau khi được sinh ra. Với việc định nghĩa ra luồng xử lý CI cho hệ thống CI/CD ở phần trước, thì ở phần này, chúng ta cần thiết lập 2 Jenkins Job sau để xây dựng và thực thi 2 luồng xử lý trên:

- Jenkins Job **CI_dev** : Lắng nghe và xử lý event Open Merge Request event và Merge Request Note event - Rebuild Merge Request event khi có sự kiện merge request build từ nhánh phát triển tính năng tới nhánh phát triển chung => Jenkins Job này thực thi luồng xử lý Open Merge Request Build và ứng dụng được deploy tới môi trường test.

- Jenkins Job **CI_staging** : Lắng nghe và xử lý event Open Merge Request event và Merge Request Note event - Rebuild Merge Request event khi có sự kiện merge request build từ nhánh phát triển chung tới nhánh chính để chuẩn bị



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 14/49

release chức năng=> Jenkins Job này thực thi luồng xử lý Open Merge Request Build và ứng dụng sẽ được deploy tới môi trường staging.

Với mỗi Jenkins Job, chúng ta cần thực hiện các cấu hình sau:

Bước 1: Tạo một folder cho Project. Folder này sẽ chứa tất cả các job triển khai CI/CD của project đó. → **NOTE:** Folder chỉ tạo một lần. Mỗi một project chỉ có 1 folder duy nhất trên Jenkins.

Enter an item name

thinghub **1**

» Required field

- Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Pipeline**
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
- External Job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.
- Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Bitbucket Team/Project**
Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.
- Folder** **2**
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.
- Organization** **3**
GitHub organization (or user account) for all repositories matching some defined markers.

OK

Mã văn bản: HD.VTS.GP.12

Số văn bản: 12

Ngày ban hành: 14/02/2020



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 15/49

Bước 2: Tạo job CI tương ứng.

Job CI, CD, rollback được tạo trong folder. Để tạo một job build jenkins tương ứng với project thực hiện vào folder. Folder mới tạo sẽ có button **create new job** như hình dưới đây:

Tạo job CI_dev như sau:

Mã văn bản: HD.VTS.GP.12

Số văn bản: 12

Ngày ban hành: 14/02/2020



Hãy nói theo cách của bạn

**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 16/49

Bước 3: Cấu hình lựa chọn các Gitlab event mà job sẽ xử lý

Mã văn bản: HD.VTS.GP.12

Số văn bản: 12

Ngày ban hành: 14/02/2020



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 17/49

Bước 4: Cấu hình init script

Cả 2 job đều dùng chung 1 Init Script: Do hạn chế của GitLab plugin, nên cần thiết lập script tiền xử lý lấy Jenkinsfile chứa script xử lý sự kiện. Script tiền xử lý này để ở phần Pipeline Script trong cấu hình của job. Ví dụ, đoạn script tiền xử lý sau thực hiện lấy Jenkinsfile của merge request rồi thực thi các xử lý đã được định nghĩa trong Jenkinsfile:

```
node('slave_43'){
    checkout changelog: true, poll: true, scm: [
        $class : 'GitSCM',
        branches : [[name: "dev_cicd"]],
        doGenerateSubmoduleConfigurations: false,
        extensions : [[class: 'UserIdentity', email:
'hienptt22@viettel.com.vn', name: 'hienptt22']],
        submoduleCfg : [],
        userRemoteConfigs : [[credentialsId: '63265de3-8396-40f9-803e-
5cd0b694e519',
                                name : 'origin',
                                url : "${env.gitlabSourceRepoHomepage}" +
".git"]]
    ]
    jenkinsfile_bootstrap = load 'jenkinsfile_bootstrap.groovy'
    jenkinsfile_bootstrap.bootstrap_build()
}
```




**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 18/49

Đoạn script trên thực thi các công việc sau:

- Lựa chọn slave node có label là slave_43 để làm build node cho build này. Các thực thi trong các script sau này sẽ được thực thi ở build node này, trừ trường hợp người thiết lập CI/CD chuyển hướng luồng xử lý sang build node khác bằng cách sử dụng keyword node.

- Checkout source code ở nhánh dev_cicd về build node. Sau đó load các CI/CD script jenkinsfile_bootstrap.groovy và Jenkinsfile khác.

- Luồng thực thi nhảy tới thực thi phương thức bootstrap_build được định nghĩa trong file jenkinsfile_bootstrap, quá trình xử lý 1 CI/CD event được bắt đầu.

Mục đích của việc load các CI/CD script từ nhánh dev_cicd (nhánh này tương ứng với nhánh phát triển chung, nhằm đảm bảo rằng mọi thay đổi tới luồng xử lý CI/CD của môi trường **staging** được kiểm soát.

Trong trường hợp job **CI_staging** thực hiện cấu hình trên Jenkins tương tự CI_dev tuy nhiên phần **Allows Branch** chỉ include branch master và phần init script sẽ load CI/CD script từ nhánh master, nhằm đảm bảo rằng mọi thay đổi tới luồng CI/CD của môi trường **productions** được kiểm soát bởi maintainer vì mã nguồn trong master branch trên repository chỉ có thể được cập nhật bởi maintainer của dự án.

NOTE: Init script của vùng staging sẽ chỉnh sửa phần bôi đỏ của init script CI_dev với branch là master.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 19/49

4.1.2. Cấu hình gitlab

Sau khi đã thiết lập Jenkins Jobs bên phía hệ thống Jenkins, chúng ta cần thiết lập các webhook để khi có một sự kiện xảy ra trên Gitlab Project, GitLab sẽ gửi các sự kiện này tới đúng các Jenkins Job chịu trách nhiệm xử lý loại sự kiện đó. Thực hiện tạo các webhook bằng cách vào phần **Settings** của dự án -> mục **Integration**.

Với mỗi job được tạo trên Jenkins, thực hiện thêm 1 webhook, ở mỗi webhook cấu hình hai thông tin: Địa chỉ của jenkins job xử lý webhook này, và loại event mà Webhook sẽ gửi đi.

4.1.3. Thiết lập các luồng xử lý tích hợp liên tục – CI pipelines

Luồng xử lý CI/CD được thiết lập bằng cách viết Jenkins pipeline script. Các luồng xử lý đã được nêu ra sẽ dùng chung một pipeline script để dễ quản lý cũng như cho phép các luồng xử lý dùng chung các thành phần giống nhau (các stage, các phương thức được dùng trong nhiều luồng xử lý).

Trước mỗi luồng xử lý CI/CD sẽ có một hình vẽ minh họa các stage mà luồng xử lý đó sẽ đi qua. Hiện tại các stage này đang được sắp xếp theo một đường thẳng, stage nào viết trước hay được gọi trước sẽ chạy trước, và kết quả chạy của stage chạy trước sẽ ảnh hưởng tới việc stage sau có được chạy hay không.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 20/49

Tuy nhiên khi áp dụng các thiết kế luồng xử lý này vào 1 project, tùy thuộc vào nhu cầu mà người quản trị CI/CD có thể thực hiện

Để phân biệt các luồng xử lý, chúng ta dựa trên các biến môi trường được Gitlab Plugin gửi lên để xác định loại event nào đã trigger build này. Dựa vào thông tin này chúng ta sử dụng rẽ nhánh if/else để đưa luồng thực thi đến đúng luồng xử lý của loại sự kiện đó.

4.1.3.1. Xác định loại event trigger build và rẽ nhánh build tới đúng luồng xử lý

Như đã nói ở trên, do chúng ta sử dụng chung một CI/CD script để xử lý tất cả các luồng, các sự kiện CI/CD. Vì vậy, ở bước đầu tiên luồng thực thi script, chúng ta cần xác định xem sự kiện kích hoạt luồng xử lý thuộc loại sự kiện nào, từ đó điều hướng luồng xử lý đến đúng script xử lý loại sự kiện được gửi tới.

Việc xác định loại sự kiện của sự kiện kích hoạt được thực hiện thông qua việc kiểm tra các biến môi trường được Jenkins GitLab plugin inject vào build, bằng phương thức **checkBuildType()**:

```
def checkBuildType() {
    def buildType = "none"
    if ("${env.gitlabActionType}".toString() == "PUSH") {
        buildType = "push_commit_build"
    } else if ("${env.gitlabActionType}".toString() == "MERGE") {
        if ("${env.gitlabMergeRequestState}".toString() == "opened") {
            buildType = "merge_request_build"
        } else if ("${env.gitlabMergeRequestState}".toString() == "closed") {
            buildType = "close_mr_build"
        } else if ("${env.gitlabMergeRequestState}".toString() == "merged") {
            buildType = "accept_mr_build"
        } else {
            buildType = "merge_request_build"
        }
    } else if ("${env.gitlabActionType}".toString() == "NOTE") {
        buildType = "rebuild_merge_request"
    } else if ("${env.gitlabActionType}".toString() == "TAG_PUSH") {
        buildType = "deploy_production"
    }
}
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 21/49

}

return buildType

}

Sau khi đã lấy được loại sự kiện bằng phương thức **checkBuildType()**, script thực hiện điều hướng luồng xử lý tới đúng script xử lý loại sự kiện này. Phương thức **bootstrap_build** được sử dụng để xử lý đúng loại sự kiện.

```
def bootstrap_build() {  
    initGlobalEnv()  
    env.BUILD_TYPE = checkBuildType()  
    switch (env.BUILD_TYPE) {  
        case "push_commit_build":  
            echo "push_commit_build"  
            bootstrapPushCommitBuild()  
            break  
        case "merge_request_build":  
            bootstrapMergeRequestBuild()  
            echo "merge_request_build"  
            break  
        case "rebuild_merge_request":  
            bootstrapRebuildMergeRequest()  
            echo "rebuild_merge_request"  
            break  
        case "accept_mr_build":  
            break  
        case "close_mr_build":  
            bootstrapAcceptAndCloseMergeRequestBuild()  
            echo "close_mr_build"  
            break  
        case "deploy_production":  
            bootstrapDeployToProduction()  
            echo "deploy_production"  
            break  
        default:  
            break  
    }  
}
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 22/49

4.1.3.2. Thiết lập luồng xử lý Merge Request Build (Open Merge Request và Rebuild Merge Request)

Các script được định nghĩa trong các file groovy cụ thể như sau:

- Một số hàm dùng chung được viết vào file **jenkinsfile_utils.groovy** để có thể dùng chung.
- **jenkinsfile_bootstrap.groovy** được sử dụng để định nghĩa việc xử lý các event trên gitlab và luồng xử lý.
- **jenkinsfile_CI.groovy** được sử dụng để định nghĩa luồng xử lý CI.
- **jenkinsfile_CD.groovy** được sử dụng để định nghĩa luồng xử lý CD và rollback.

Một số stage được sử dụng trong luồng **CI_dev** và **CI_staging** cụ thể như sau:

a. Stage checkoutSourceCode

```
def checkoutSourceCode(checkoutType){
    if (checkoutType == "PUSH"){
        checkout changelog: true, poll: true, scm: [
            $class
                : 'GitSCM',
            branches
                : [[name: "${env.gitlabAfter}"]],
            doGenerateSubmoduleConfigurations: false,
            extensions
                : [$class: 'UserIdentity',
                    email : 'hienptt22@viettel.com.vn', name:
'hienptt22'],
                [$class: 'CleanBeforeCheckout']],
            submoduleCfg
                : [],
            userRemoteConfigs
                : [[credentialsId: "63265de3-8396-
40f9-803e-5cd0b694e519",
                    url
                        :
"${env.gitlabSourceRepoHomepage}" + ".git"]]
        ]
    }
}
```

Ở luồng push commit build, stage này thực hiện việc kết nối tới Gitlab (env.gitlabSourceRepoHomepage) và lấy mã nguồn tại commit có commitID tương ứng event push commit env.gitlabAfter trong Gitlab repository.

Ở script này, thông tin xác thực được sử dụng để kết nối với GitLab là credentialsId: **"\${env.GitSCM_CredentialsId}"**. Thông tin xác thực này là thông tin username và password của một GitLab Account có quyền truy cập vào


**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 23/49

project được build. Credentials này được lưu dưới dạng username-password vì chúng ta sử dụng giao thức http để checkout repository.

Checkout Merge Request Build:

Bước này thực hiện checkout source code trước khi build. Khác với Push commit build, để checkout source code cho Merge Request Build, chúng ta phải xác định source branch và target branch của Merge Request kích hoạt Build, sau đó thực hiện merge source branch vào target branch.

```
else if (checkoutType == "MERGE") {
  checkout changelog: true, poll: true, scm: [
    $class : 'GitSCM',
    branches : [[name: "origin/${env.gitlabSourceBranch}"]],
    doGenerateSubmoduleConfigurations: false,
    extensions : [[ $class : 'PreBuildMerge',
      options: [
        fastForwardMode: 'FF',
        mergeRemote : 'origin',
        mergeStrategy : 'RESOLVE',
        mergeTarget : "${env.gitlabTargetBranch}"
      ]
    ]],
    [$class: 'UserIdentity',
      email : 'hienptt22@viettel.com.vn', name:
'hienptt22'],
    [$class: 'CleanBeforeCheckout']],
    submoduleCfg : [],
    userRemoteConfigs : [[credentialsId: "63265de3-8396-40f9-
803e-5cd0b694e519",
url : "${env.gitlabSourceRepoHomepage}" +
".git"']]
  ]
}
```

Ở bước merge source branch vào target branch, chúng ta có 3 chế độ merge có thể lựa chọn :

- **--ff**: When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 24/49

- **--no-ff**: Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag that is not stored in its natural place in refs/tags/ hierarchy.

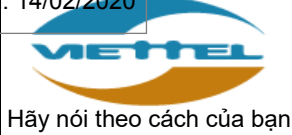
- **--ff-only**: Refuse to merge and exit with a non-zero status unless the current HEAD is already up to date or the merge can be resolved as a fast-forward.

b. Stage Unit Test

Stage này thực hiện chạy Unit Test đã viết và tính Coverage Code stage('3.1. Unit Test & Code Coverage'){

```
try {
    echo "code coverage started"
    sh './build.sh coverage'
    echo "code coverage done"
    jacoco([
        classPattern: 'ms-device/target/classes,auth/target/classes,ms-user-
manage/target/classes,ms-app-manage/target/classes',
        sourcePattern: 'ms-device/src/main/java,auth/src/main/java,ms-user-
manage/src/main/java,ms-app-manage/src/main/java'
    ])
    def coverageResultStrComment = "<b>Coverage Test Result:</b>"
    <br/><br/>
    def coverageInfoXmlStr = readFile "cicd/target/jacoco-aggregate-
report/jacoco.xml"
    echo "Coverage Info:
${getProjectCodeCoverageInfo(coverageInfoXmlStr)} "
    coverageResultStrComment +=
getProjectCodeCoverageInfo(coverageInfoXmlStr)
    coverageResultStrComment += "<i><a
href='${env.BUILD_URL}Code-Coverage-Report/jacoco>' +
'Details Code Coverage Test Report...</a></i><br/><br/>"
    env.CODE_COVERAGE_RESULT_STR =
coverageResultStrComment
} catch (err) {
    echo "Error when test Unit Test"
    throw err
}
```

Sau đó ghi nhận lại kết quả và đẩy kết quả lên Jenkins thông qua lệnh DSL **junit** và sau đó truy cập vào Jenkins lấy kết quả build thông qua phương thức helper **getUnitTestStatus** để add kết quả chạy unit test vào GitLab Build Result Comment ở block **finally**



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 25/49

```
finally {
    sh 'ls -al'
    junit '*/target/*-results/test/TEST-*.xml'
    def unitTestResult = getTestResultFromJenkins()
    env.UNIT_TEST_PASSED = unitTestResult["passed"]
    env.UNIT_TEST_FAILED = unitTestResult["failed"]
    env.UNIT_TEST_SKIPPED = unitTestResult["skipped"]
    env.UNIT_TEST_TOTAL = unitTestResult["total"]
    def testResultContent = "- Passed: <b>${unitTestResult['passed']}</b>
<br/>" +
        "- Failed: <b>${unitTestResult['failed']}</b> <br/>" +
        "- Skipped: <b>${unitTestResult['skipped']}</b> <br/>"
    def testResultString = "<b>Unit Test Result:</b>
<br/><br/>${testResultContent}" +
        "<i><a href='${env.BUILD_URL}testReport/'>Details Unit Test
Report...</a></i><br/><br/>"
    env.UNIT_TEST_RESULT_STR = testResultString
    if (unitTestResult['failed'] > 0) {
        error "Failed ${unitTestResult['failed']} unit tests"
        env.UNIT_TEST_RESULT_STR += "Failed
${unitTestResult['failed']} unit tests"
    }
}
```

c. Stage SonarQube

Stage này thực hiện quét SonarQube cho source code của project để phát hiện các lỗi, các vi phạm An toàn thông tin tiềm ẩn. Để kết nối với SonarQube Server, người thiết lập CI/CD cần sử dụng block DSL **withSonarQubeEnv**, với tham số truyền vào là một trong số các SonarQube server được cấu hình trên Jenkins, cần liên lạc với team quản trị hệ thống Jenkins để lấy danh sách các SonarQube server khả dụng.

Tool được sử dụng trong stage này là sonarqube plugin của maven. Trước khi quét sonarQube, cần sinh ra sonarQube key được sử dụng trong lần quét này. Hiện tại, các project trong tài liệu sinh SonarQube key theo tên Project và tên nhánh source/ target branch của sự kiện trigger build.


**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 26/49

```

stage('3.2. SonarQube analysis') {
    env.SONAR_CUBE_PROJECT_KEY = genSonarQubeProjectKey()
    withSonarQubeEnv('SONARQ_V6'){
        sh "/home/app/server/sonar-scanner/bin/sonar-scanner " +
            "-Dsonar.projectName=${env.SONAR_CUBE_PROJECT_KEY} " +
            "-Dsonar.projectKey=${env.SONAR_CUBE_PROJECT_KEY} " +
            "-Dsonar.java.binaries=. " +
            "-Dsonar.sources=./ " +
            "-Dsonar.inclusions=**/*.java,**/*.cs " +
            "-Dsonar.exclusions=**/res/**,**/target/**,**/build/**,**/share/**,**.html*,*/.settings/**,**/.mvn/**"
        sh 'ls -al'
        sh 'cat .scannerwork/report-task.txt'
        def props = readProperties file: '.scannerwork/report-task.txt'
        env.SONAR_CE_TASK_ID = props['ceTaskId']
        env.SONAR_PROJECT_KEY = props['projectKey']
        env.SONAR_SERVER_URL = props['serverUrl']
        env.SONAR_DASHBOARD_URL = props['dashboardUrl']
        echo "SONAR_SERVER_URL: ${env.SONAR_SERVER_URL}"
        echo "SONAR_PROJECT_KEY: ${env.SONAR_PROJECT_KEY}"
        echo "SONAR_DASHBOARD_URL: ${env.SONAR_DASHBOARD_URL}"
    }
}

```

d. Stage Quality Gate

Sau khi thực hiện quét xong Maven SonarQube ở máy Build Machine, cần chờ cho đến khi SonarQube server xử lý xong. Ở các project trong tài liệu, hệ thống thực hiện giai đoạn này bằng một vòng lặp timeout, nếu chờ quá 10 lần mà không nhận được kết quả, build sẽ bị đánh dấu là fail. Nếu như SonarQube server trả về kết quả fail, build cũng bị tính là fail.

```

stage("3.3. Quality Gate") {
    def qg = null
    try {
        def sonarQubeRetry = 0

```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 27/49

```
def sonarScanCompleted = false
while (!sonarScanCompleted) {
    try {
        sleep 10
        timeout(time: 1, unit: 'MINUTES') {
            script {
                qg = waitForQualityGate()
                sonarScanCompleted = true
                if (qg.status != 'OK') {
                    if (env.bypass == 'true') {
                        echo "Sonar contain error"
                    } else {
                        error "Pipeline failed due to quality gate failure:
${qg.status}"
                    }
                }
            }
        }
    } catch (FlowInterruptedException interruptEx) {
        // check if exception is system timeout
        if (interruptEx.getCauses()[0] instanceof
org.jenkinsci.plugins.workflow.steps.TimeoutStepExecution.ExceededTimeout)
        {
            if (sonarQubeRetry <= 10) {
                sonarQubeRetry += 1
            } else {
                if (env.bypass == 'true') {
                    echo "Sonar contain error"
                } else {
                    error "Cannot get result from Sonarqube server. Build
Failed."
                }
            }
        } else {
            throw interruptEx
        }
    }
}
```




**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 28/49

```

    }
  }
  catch (err) {
    throw err
  }
}
}
catch (err) {
  throw err
} finally {
  def
                                codeAnalysisResult
getSonarQubeAnalysisResult(env.SONAR_SERVER_URL,
env.SONAR_PROJECT_KEY)
  def sonarQubeAnalysisStr = "- Vulnerabilities:
<b>${codeAnalysisResult["vulnerabilities"]}</b> <br/>" +
  "- Bugs: <b>${codeAnalysisResult["bugs"]}</b> <br/>" +
  "- Code Smell: <b>${codeAnalysisResult["code_smells"]}</b>
<br/>"
  def sonarQubeAnalysisComment = "<b>SonarQube Code Analysis
Result: ${qg.status}</b> <br/><br/>${sonarQubeAnalysisStr} " +
  "<i><a href='${SONAR_DASHBOARD_URL}'>" +
  "Details SonarQube Code Analysis Report...</a></i><br/><br/>"
  env.SONAR_QUBE_SCAN_RESULT_STR
sonarQubeAnalysisComment
  if ("${env.gitlabActionType}".toString() == "MERGE" ||
"${env.gitlabActionType}".toString() == "NOTE") {
    echo "check vulnerabilities, code smell and bugs"
    int maximumAllowedVulnerabilities
env.MAXIMUM_ALLOWED_VUNERABILITIES as Integer
    int maximumAllowedBugs = env.MAXIMUM_ALLOWED_BUGS
as Integer
    int maximumAllowedCodeSmell
env.MAXIMUM_ALLOWED_CODE_SMELL as Integer
    echo "maximum allow vulnerabilities:
${maximumAllowedVulnerabilities} "
    echo "maximum allow bugs: ${maximumAllowedBugs}"
    echo "maximum allow code smell: ${maximumAllowedCodeSmell}"
    if (codeAnalysisResult["vulnerabilities"] >
maximumAllowedVulnerabilities ||

```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 29/49

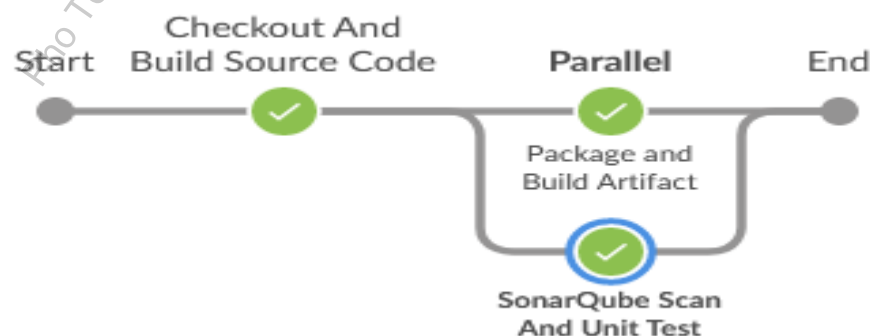
```
codeAnalysisResult["bugs"] > maximumAllowedBugs ||
codeAnalysisResult["code_smells"] > maximumAllowedCodeSmell) {
    if (env.bypass == 'true') {
        echo "Vulnerability, code smell or bug number overs allowed
limits!"
    } else {
        error "Vulnerability, code smell or bug number overs allowed
limits!"
    }
}
}
```

Cuối cùng, kết quả SonarQube được lấy thông qua câu lệnh: `def codeAnalysisResult = getSonarQubeAnalysisResult(env.SONAR_SERVER_URL, env.SONAR_PROJECT_KEY)` sau đó kết quả quét sonarqube được kiểm tra, nếu số lượng lỗi hoặc số lượng nguy cơ an toàn thông tin vượt ngưỡng, build bị đánh dấu là fail.

4.1.3.3. Thiết lập luồng xử lý Push Commit Build

Ngoài ra, khi muốn test nhanh kết quả quét sonar hay kiểm tra unit test đối với một số case đơn giản, có thể thực hiện trigger push commit để kiểm tra nhanh. Do dev thường xuyên tạo một push event tới gitlab nên để giảm tải cho hệ thống CI/CD đồng thời tránh việc những thay đổi trong push event chưa cần đẩy lên server test luôn thì có thể cấu hình chỉ quét sonar và chạy unittest và build khi có commit code để tránh trường hợp build failed khi merge commit.

Luồng xử lý push commit build:





**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

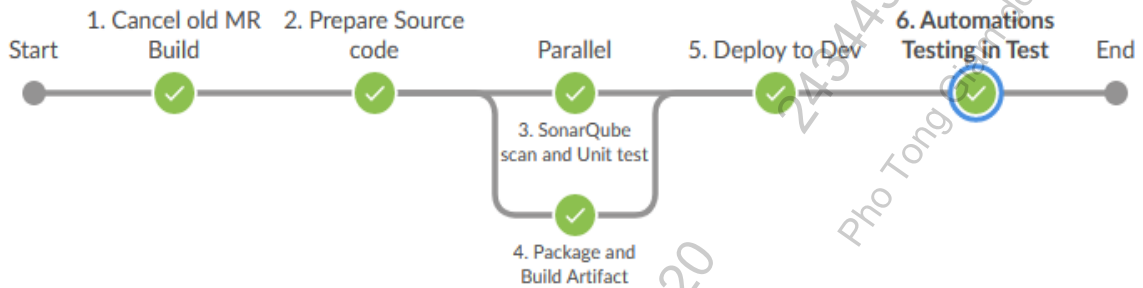
Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 30/49

4.1.3.4. Cancel Old Merge Request Build - Thiết lập cơ chế cancel các build không cần thiết trong quá trình build một Merge Request

Thiết lập luồng xử lý Merge Request Build (Open Merge Request và Rebuild Merge Request)



Trước khi thực hiện các tác vụ chính của luồng xử lý Merge Request Build, luồng xử lý Merge Request Build cần phải thực hiện việc hủy các luồng xử lý Build Trước đó nằm trên cùng Merge Request này, để nhằm:

- Giảm tải cho hệ thống CI/CD: Hủy các build không còn cần thiết nữa, vì việc các build này có chạy tiếp hay không ảnh hưởng tới kết quả cuối cùng của Merge Request Build. Ngừng chúng đi giúp hệ thống CI/CD được giảm tải vì không phải thực thi Các build không có ý nghĩa

- Tránh xung đột tài nguyên giữa các Build trong cùng Một Request, đảm bảo điều kiện trong một thời điểm chỉ có một Merge Request Build tác động và thay đổi môi trường Test hoặc Staging của Merge Request. Điều này cho phép giữ tính đúng đắn của môi trường Staging và Test, đảm bảo phần mềm chạy trên môi trường Test hoặc Staging là phần mềm được build từ source code mới nhất của Open Merge Request.

Cơ chế **Cancel Old Merge Request Build** được implement như sau:

```
def cancelOldMrBuild(gitlabMergeRequestId, currentBuildType) {  
  env.CANCEL_BUILD_WARNING = ""  
  withCredentials([  
    usernamePassword(credentialsId:  
      '63265de3-8396-40f9-803e-5cd0b694e519', usernameVariable: 'username', passwordVariable: 'password'),  
    string(credentialsId: "jenkins-api-token", variable: 'jenkins_api_token')]) {  
    def pipelines = httpRequest([  
      acceptType : 'APPLICATION_JSON',  
      httpMode : 'GET',  
    ])
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 31/49

```

contentType : 'APPLICATION_JSON',
customHeaders: [[name: 'Private-Token', value: password]],
url
:
"${env.GITLAB_PROJECT_API_URL}/merge_requests/${gitlabMergeReques
tId}/pipelines"
})
for (pipeline in jenkinsfile_utils.jsonParse(pipelines.content)) {
//noinspection GroovyAssignabilityCheck
def checkCommitID = pipeline['sha']
echo "check commit id: ${checkCommitID}"
def commitJobs = httpRequest([
    acceptType : 'APPLICATION_JSON',
    httpMode : 'GET',
    contentType : 'APPLICATION_JSON',
    customHeaders: [[name: 'Private-Token', value: password]],
    url
:
"${env.GITLAB_PROJECT_API_URL}/repository/commits/${checkCommitID}
D}/statuses?all=yes"
])
for (commitJob in jenkinsfile_utils.jsonParse(commitJobs.content)) {
//noinspection GroovyAssignabilityCheck
if (currentBuildType == "merge_request_build" || currentBuildType ==
"rebuild_merge_request"
|| ((currentBuildType == "accept_mr_build" || currentBuildType ==
"close_mr_build")
&& (commitJob["name"].contains(env.MR_BUILD_PREFIX)
||
commitJob["name"].contains(env.ACCEPT_CLOSE_MR_BUILD_PREFIX)))
){
    if (commitJob["status"] == "pending" || commitJob["status"] ==
"running") {
        def buildURL = commitJob["target_url"].substring(0,
commitJob["target_url"].length() - 17)
        echo "Check buildURL: ${buildURL}"
        echo "Current buildURL: ${env.BUILD_URL}"
    }
}
}

```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 32/49

```

if (!env.BUILD_URL.contains(buildURL)) {
    def retry = 0
    while (checkIfBuildIsRunning(buildURL) && retry < 3) {
        echo "Old build: ${commitJob["target_url"]} is running!. Stop
this job!"

        httpRequest([
            acceptType : 'APPLICATION_JSON',
            httpMode   : 'POST',
            contentType : 'APPLICATION_JSON',
            customHeaders: [[name: 'Authorization', value:
jenkins_api_token]],
            url         : "${buildURL}/stop"
        ])
        sleep 10
        retry += 1
    }
    if (checkIfBuildIsRunning(buildURL)) {
        env.CANCEL_BUILD_WARNING += "<h2> Build
${buildURL} is still running after cancel build 3 times. Re check it!</h2>"
    }
}
}
}
}
}

```

Phương thức **cancelOldMrBuild** thực hiện việc sau:

- Kết nối tới GitLab thông qua API để lấy ra danh sách các build pipeline đang chạy trên merge request mà build hiện tại đang xử lý
- Kiểm tra từng build pipeline, lấy ra danh sách build status của commit tương ứng với pipelines đó.
 - Kiểm tra từng build status, xác định xem build status đó còn đang running hay không thông qua Jenkins Build tương ứng với build status đó => Kiểm tra bằng Jenkins API xem build có đang running hay không.
 - Nếu Build được kiểm tra đang running và có timestamp nhỏ hơn timestamp của build hiện tại => build được kiểm tra sinh ra trước build hiện tại, thì thực hiện send cancel signal tới build được kiểm tra.



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 33/49

• Sau khi send signal, sử dụng jenkins API gửi đến link build trong build status kiểm tra lại xem build status đã thay đổi sang cancel hoặc success hoặc failure hay chưa. Nếu chưa, thực hiện retry việc cancel build này 3 lần

• Nếu sau 3 lần cancel build mà build đang check vẫn chạy, thực hiện throw lỗi: **Build is still Running, cancel build Failed.**

4.1.3.5. Kết quả cập nhật vào commit status

Để chèn build status và comment trên Gitlab ta sử dụng hàm có sẵn trong Gitlab Plugin.

Trong Gitlab Plugin có chứa một số hàm hỗ trợ trong việc update build status như gitlabCommitStatus, updateGitlabCommitStatus, gitlabBuilds. Chúng ta có thể tìm hiểu thêm tại [đây](#). Đây là phương pháp tiện lợi, code gọn gàng dễ hiểu. Tuy nhiên, do hiện tại chỉ có CI/CD Bot có quyền gọi hàm này nên để project có thể nhận được thông điệp khi gọi hàm, ta cần thêm CI/CD Bot vào danh sách member của project.

Hiện tại dự án thinghub đang sử dụng phương pháp này để hiển thị icon trạng thái build.

Để thiết lập build status (hiển thị status icon):

updateGitlabCommitStatus name: "build", state: 'running'

updateGitlabCommitStatus name: "\${env.gitlabBuildID}", state: 'running'

✚ Cập nhật build status trong Thinghub

Việc cập nhật build status trong **thinghub** xảy ra trong các trường hợp sau:

- Luồng Push Commit Build và Merge Request Build:

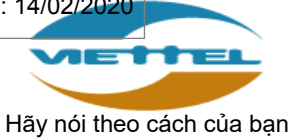
- Khi **bắt đầu chạy** luồng build: cập nhật trạng thái **running**
- Khi **bắt đầu check** kết quả build: cập nhật trạng thái **success**

- Kết thúc luồng build:

- Kết quả build là **SUCCESS**: cập nhật trạng thái **success**
- Kết quả build là **ABORTED**: cập nhật trạng thái **canceled**
- Kết quả build là **FAILURE**: cập nhật trạng thái **failed**

✚ Xác định kết quả build và cập nhật kết quả build lên GitLab và Jenkins

Sau khi luồng xử lý build thực hiện xong, hoặc đột ngột dừng lại (gặp lỗi, bị hủy...) Thì ở bước cuối cùng trong script xử lý build, chúng ta cần thực hiện việc xác định kết quả build và cập nhật kết quả build lên GitLab và Jenkins bằng cách thực hiện theo các bước sau: Ghi lại kết quả test/build của mỗi bước build



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 34/49

ra các biến môi trường như: **env.PUSH_COMMIT_BUILD_COMMENT** và **env.MERGE_REQUEST_BUILD_COMMENT**

Đối với merge request build, chúng ta sẽ tạo ra một comment ở merge request thông qua Jenkins pipeline command **addGitLabMRComment**, với kết quả build được ghi lại ở biến **buildResultContent**:

```
def buildSummary = "<summary>${buildIcon} Merge request
<strong>${display_build_name}</strong>: " +
    "${buildResultStr}</summary>"
def buildDetail = "<h4><i><a
href='${env.BUILD_URL}display/redirect'>" +
    "Build Details...</a></i></h4><br><br>"
def buildResultContent =
    (env.CANCEL_BUILD_WARNING == null ? "" :
env.CANCEL_BUILD_WARNING) +
    (env.CHECK_IF_BRANCHES_REVISION_ARE_SAME_RESULT
== null ? "" :
env.CHECK_IF_BRANCHES_REVISION_ARE_SAME_RESULT) +
    (env.UNIT_TEST_RESULT_STR == null ? "" :
env.UNIT_TEST_RESULT_STR) +
    (env.CODE_COVERAGE_RESULT_STR == null ? "" :
env.CODE_COVERAGE_RESULT_STR) +
    (env.SONAR_QUBE_SCAN_RESULT_STR == null ? "" :
env.SONAR_QUBE_SCAN_RESULT_STR) +
    (env.FUNCTIONAL_TEST_RESULT_STR == null ? "" :
env.FUNCTIONAL_TEST_RESULT_STR) +
    (env.PERFORMANCE_TEST_RESULT_STR == null ? "" :
env.PERFORMANCE_TEST_RESULT_STR)

def mergeRequestBuildStr =
    "<details> ${buildSummary}<br><br> ${buildResultContent}" +
    "${buildDetail}</details>".toString()
echo "comment ${mergeRequestBuildStr}"
addGitLabMRComment comment: "${mergeRequestBuildStr}"
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 35/49

Vì các comment trên GitLab được format bằng Markdown, nên để format kết quả build, thì ở đây chúng ta sẽ format kết quả build dưới dạng HTML code, khi đó kết quả build sẽ hiển thị trên Gitlab một cách rõ ràng và có cấu trúc:

CI-CD Bot @vtsjenkinsadmin · 1 day ago

▼ Merge request hienptt22_feature:041fe78c -> dev_cicd - MERGE: Build Success, Deploy to IP: 10.60.156.68

Unit Test Result:

- Passed: 1719
- Failed: 0
- Skipped: 0

[Details Unit Test Report...](#)

Coverage Test Result:

- INSTRUCTION: 19698/69506 (28.34%)
- BRANCH: 242/3120 (7.76%)
- LINE: 6902/18481 (37.35%)
- COMPLEXITY: 3884/7698 (50.45%)
- METHOD: 3855/6107 (63.12%)
- CLASS: 273/612 (44.61%)

[Details Code Coverage Test Report...](#)

SonarQube Code Analysis Result: ERROR

- Vulnerabilities: 56
- Bugs: 59
- Code Smell: 741

[Details SonarQube Code Analysis Report...](#)

Functional Test Result:

[Details Functional Test Report...](#)

[Build Details...](#)

4.1.3. Các vấn đề liên quan đến Jenkins credentials

4.1.3.1. Quản lý Credentials trong Jenkins

Credentials là cách mà Jenkins lưu trữ và quản lý các thông tin mật như username/password, token, ssh private key của các hệ thống liên quan đến luồng xử lý CI/CD như NexusRepository, Gitlab, SonarQube... Việc sử dụng credentials của Jenkins thay vì khai báo các thông tin mật này trong các file config cho phép bảo mật các thông tin này. Cụ thể, khi thực thi CI/CD script, nếu chúng ta sử dụng các credentials này thì giá trị của chúng không bị in trực



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020
Ngày hết hiệu lực: 01/03/2021
Lần ban hành: 01
Trang/tổng số trang: 36/49

tiếp ra console mà sẽ được che lại dưới dạng một chuỗi ****, từ đó giới hạn người dùng, chỉ những người thiết lập hệ thống CI/CD trong dự án mới có thể biết được nội dung của các thông tin mật này.

Trên hệ thống CI/CD hiện tại, mỗi một dự án có một folder-domain riêng, và các Jenkins credentials cho dự án đó cũng được khai báo trong mục Credentials của Folder đó, và chỉ khả dụng ở phạm vi các job trong folder đó hoặc trong thư mục Credential chung.

4.1.3.2. Hướng dẫn lấy credential id trong jenkins

Credential được sử dụng để checkout source code trên gitlab hoặc được sử dụng để lấy kết quả từ jenkins với jenkins-api-key.

Hướng dẫn cách lấy credential id trong jenkins để sử dụng trong phương thức checkout source code.

Trong trường hợp chưa add Credential trên jenkins, thực hiện add Credential để sử dụng như sau: truy nhập đường dẫn

http://10.60.156.96:8080/credentials/store/system/domain/_/newCredentials

để tạo mới một credential:

Kind: Username with password
Scope: Global (Jenkins, nodes, items, all child items, etc.)
Username: hienptt22
Password:
ID: hienptt22-gitlab
Description:
OK

ID được sử dụng trong các script để lấy dữ liệu trong gitlab hoặc jenkins

Trong trường hợp đã tạo credential từ trước và không **NHẬP ID** của Credential thì thực hiện các thao tác sau để lấy **credential id** để sử dụng.

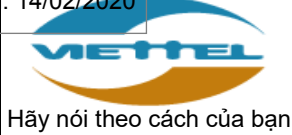
Truy cập tới link: <http://10.60.156.96:8080/credentials/>

Search: hienptt22
Highlight All Match Case Whole Words 1 of 7 matches

Icon	Name	Kind	Scope	ID	Username
Jenkins	hienptt22/*****	Username with password	Global	63265de3-8396-40f9-803e-5cd0b694e519	hienptt22/*****
Jenkins	huongnv75/*****	Username with password	Global	14b59ab4-5ad8-45c3-962c-635244e646f0	huongnv75/*****
Jenkins	huongnv75/*****	Username with password	Global	5b701045-a41b-47dc-a3c2-819efa7e0503	huongnv75/*****

Credential ID

4.1.3.3. Sử dụng jenkins credentials trong Build Script



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 37/49

Sử dụng Credential của jenkins với Username và token của gitlab để lấy source code từ gitlab thực hiện build.

checkout changelog: true, poll: true, scm: [

\$class : 'GitSCM',

branches : [[name: "\${env.gitlabAfter}"]],

doGenerateSubmoduleConfigurations: false,

extensions : [[class: 'UserIdentity',

email : 'hienptt22@viettel.com.vn', name: 'hienptt22'],

[[class: 'CleanBeforeCheckout']],

submoduleCfg : [],

userRemoteConfigs : [[credentialsId: "63265de3-8396-40f9-803e-5cd0b694e519",

url : "\${env.gitlabSourceRepoHomepage}"
+".git"]]
]

NOTE: Chỉnh sửa một số thông tin **bôi vàng ở trên** sau:

- **Email:** Email tương ứng của tài khoản gitlab
- **Name:** tên username gitlab
- **credentialsId:** ID lấy trong jenkins theo hướng dẫn ở trên.

Ngoài ra có thể sử dụng Credential đã thêm vào trong jenkins để thực hiện login tới hệ thống khác trong command line. Ví dụ như sau:

```
withCredentials([usernamePassword(credentialsId: 'docker-login',  
usernameVariable: 'username', passwordVariable: 'password')]){  
    sh """"  
    docker login -u ${username} -p ${password} 10.60.156.72  
    """"  
}
```

4.1.4. Các biến môi trường được GitLab plugin inject vào build

Khi Jenkins plugin xử lý một event, các thuộc tính của event đó sẽ được biến đổi thành các **Jenkins Environment Variable** trong **Jenkins Build** được trigger ra, các biến môi trường này sau đó có thể được sử dụng trong CI/CD script để:



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 38/49

- Xác định loại event đang được xử lý => Rẽ nhánh luồng xử lý đến đúng luồng xử lý của từng loại sự kiện
- Xác định các nhánh source code liên quan đến event => checkout và merge source code chính xác

Danh sách các biến môi trường được Jenkins plugin inject vào được liệt kê ở đây:

gitlabBranch
gitlabSourceBranch
gitlabActionType
gitlabUserName
gitlabUserEmail
gitlabSourceRepoHomepage
gitlabSourceRepoName
gitlabSourceNamespace
gitlabSourceRepoURL
gitlabSourceRepoSshUrl
gitlabSourceRepoHttpUrl
gitlabMergeRequestTitle
gitlabMergeRequestDescription
gitlabMergeRequestId
gitlabMergeRequestIid
gitlabMergeRequestState
gitlabMergedByUser
gitlabMergeRequestAssignee
gitlabMergeRequestLastCommit
gitlabMergeRequestTargetProjectId
gitlabTargetBranch
gitlabTargetRepoName
gitlabTargetNamespace
gitlabTargetRepoSshUrl
gitlabTargetRepoHttpUrl
gitlabBefore
gitlabAfter
gitlabTriggerPhrase

4.1.5. Vấn đề về build node và node label



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 39/49

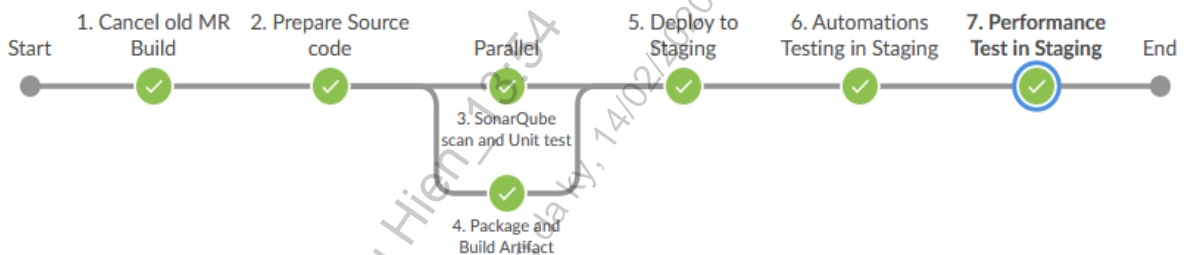
Trên một hệ thống Jenkins có thể tồn tại nhiều loại build node khác nhau, mỗi loại build node có một label riêng là tên của loại build node đó. Mặc định thì khi sử dụng init script ở phần Cấu hình Jenkins, loại build node **slave_43** được sử dụng, loại build node này HĐH là centos 7, và được cài sẵn các tool như maven, ansible, node_modules, ant,...

4.1.6. Song song hóa các luồng thực thi để giảm thời gian build

Để giảm thời gian build một sự kiện, chúng ta có thể sử dụng Jenkins Parallel để các công việc trong cùng 1 giai đoạn mà không có mối quan hệ phụ thuộc với nhau có thể thực hiện cùng 1 lúc. Ví dụ trong luồng xử lý merge request build, các công việc:

- Unit Test, Code Coverage & SonarQube Scan
- Build, package và upload to repository

Có thể thực hiện song song với nhau, từ đó giảm thời gian build



Các luồng parallel chạy trên các node khác nhau:

```
def tasks = [:]
tasks['3. SonarQube scan and Unit test'] = {
    node("slave_43") {
        sonarQubeScanAndUnitTest("MERGE")
    }
}

tasks['4. Package and Build Artifact'] = {
    node("slave_43") {
        buildService("MERGE", serviceList)
    }
}

parallel tasks
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 40/49

Điểm khác biệt giữa parallel ở các node khác nhau, đó là ở đầu luồng parallel, nếu luồng xử lý có sử dụng đến source code, thì chúng ta cần checkout lại source code, vì khi dùng block **node**, tức là chúng ta sẽ thực hiện luồng build ở node mới khác node hiện tại, và khi bắt đầu chúng ta sẽ không có source code ở node mới.

4.1.7. Project Layout

Tùy thuộc vào ngôn ngữ và kiến trúc dự án, mà mỗi dự án phần mềm có thể có một layout riêng. Layout của một dự án ảnh hưởng tới việc thiết lập các script CI/CD cho dự án, ví dụ như khi viết các stage unit test, hoặc compile, có thể dựa vào project layout để chỉ định chính xác module nào cần test, compile, hoặc dựa vào project layout để build song song nhiều module cùng một lúc, giúp giảm thời gian thực thi luồng xử lý CI/CD.

Trong phần này, sẽ giới thiệu về project layout của dự án **thinghub** sau khi áp dụng CI/CD.

- Module cicd: Là module thực hiện nhiệm vụ aggregate code coverage của các service để tính ra code coverage chung của dự án.

- File pom ở thư mục gốc của dự án có vai trò khai báo các sub module của dự án:

```
<modules>
  <module>auth</module>
  <module>ms-device</module>
  <module>ms-user-manage</module>
  <module>ms-app-manage</module>
  <module>service-discovery</module>
  <!-- CI/CD module -->
  <module>cicd</module>
</modules>
```

Tham khảo file config tại [đây](#).

4.1.8. Hướng dẫn cấu hình tính Coverage Code qua jenkins.

Để có thể tính Coverage Code sau khi chạy xong Unit Test. Thực hiện thêm trong file pom.xml của từng module có viết Unit Test phần code sau để có thể tích hợp vào luồng CI/CD trên jenkins.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

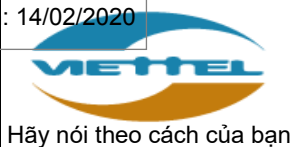
Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 41/49

```
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.2</version>
  <executions>
    <execution>
      <id>prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
    <execution>
      <id>post-unit-test</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
      <configuration>
        <dataFile>target/jacoco.exec</dataFile>
        <!-- Sets the output directory for
the code coverage report. -->
        <outputDirectory>target/jacoco-
aggregate-report</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
<configuration>
  <excludes>
    <exclude>jdk.internal.*</exclude>
  </excludes>
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 42/49

</configuration>

</plugin>

</plugins>

</build>

Tham khảo file cấu hình tại [đây](#):

Để tích hợp tính kết quả Coverage Code cho tất cả các module để thực hiện cấu hình unittest, tạo một folder cicd trong project với một file pom.xml. File pom.xml sẽ cấu hình các **dependency** là các file build artifact của các module có viết unittest.

Tham khảo cấu hình tại [đây](#).

4.1.9. Hướng dẫn sử dụng ansible trong quá trình auto deploy ứng dụng.

Ansible là công cụ tự động khá hiệu quả trong việc triển khai ứng dụng tới nhiều server một cách nhanh gọn và tiện lợi.

Trong quá trình phát triển phần mềm có triển khai CI/CD, ansible được áp dụng trong stage deploy ứng dụng trên các môi trường khác nhau, đảm bảo việc triển khai ứng dụng được nhanh gọn và đảm bảo thực hiện đầy đủ các bước được yêu cầu.

Trong pipeline deploy ứng dụng, sử dụng stage để định nghĩa các thao tác chạy như sau:

```
stage("Deploy to production with version build number $version"){
    echo "Build version $version"
    node('slave_43'){
        dir('/u01/jenkins/workspace/TCQG/CD'){
            sh "sudo ansible-playbook deploy_to_productions_test.yml -
e VERSION=${version} -e GROUPID=TCQG_CI_Web -vvv"
        }
    }
}
```

Stage này chỉ định sử dụng playbook để deploy ứng dụng qua jenkins. Tất cả các thao tác cần thực hiện khi deploy ứng dụng được định nghĩa trong playbook deploy_to_productions_test.yml

Để hiểu chi tiết về ansible, các develop có thể tham khảo hướng dẫn tại [đây](#).

4.1.10. Các kiến thức và công nghệ chính đã được sử dụng để thiết lập luồng CI/CD

Để có thể xây dựng và maintain luồng CI/CD cho hệ thống thinghub, đã sử dụng các kiến thức và công nghệ chính sau:

- Sơ đồ hệ thống CI/CD, thành phần và vai trò của các thành phần trong hệ thống.


**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 43/49

- Ngôn ngữ để viết Jenkins pipelines: Một Groovy DSL có tên là Jenkins Scripted pipeline

- Note: Jenkins pipeline syntax main document:

<https://jenkins.io/doc/book/pipeline>

- Các thiết lập cần thiết cho CI/CD bên GitLab và Jenkins:

- Kết nối
- Job
- Jenkins Credentials
- Các event từ GitLab gửi lên

- Cách thiết lập một jenkins job để build 1 GitLab event.

- Các bước cơ bản của một push commit build, một merge request build.

- Các công cụ cần thiết, các tool cần thiết để viết các stage trong build.

- Các kết nối, API, các hệ thống bên cạnh Jenkins trong môi trường CI/CD:

- SonarQube API
- GitLab API
- Jenkins API

- Các công cụ để tương tác với các Hệ thống và các môi trường:

- HTTP request
- Ansible.

- Các tính năng nâng cao để tối thiểu thời gian build: Parallel

- Quản lý dự án và repository với git.

- Git - mô hình phân nhánh, nhánh trên repo và nhánh ở local.

- Maven & Maven multi module

- Jacoco Code coverage

4.1.11. Một số lưu ý

- Script được sử dụng chưa áp dụng một số tính năng nâng cao của Jenkins như parallel. Nếu cần giảm thời gian build hơn nữa, chúng ta có thể sử dụng tính năng này trong luồng xử lý.

- Không tạo ra 2 merge request có cùng source branch, do hạn chế của Gitlab là kết quả của merge request build trên Gitlab là kết quả của commit cuối cùng trên source branch => khi để 2 merge request cùng source branch dễ gây ra xung đột giữa các build của 2 merge request.

- Khi tạo job để test CI/CD script mới, cần cấu hình cẩn thận để test job không build nhầm các nhánh và commit production

4.2. Hướng dẫn CD pipeline cho một dự án cụ thể



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 44/49

Trong phần này sẽ trình bày các bước để thiết lập một luồng CD hoàn chỉnh, xuyên suốt phần này, các stage của một luồng CD sẽ được viết theo trình tự thực hiện trong một luồng CD.

Stage Checkout Source Code: ở stage này, sau khi bắt được event đánh tag để release version trên gitlab, jenkins sẽ thực hiện lấy source code trên gitlab về để chạy các script.

```
stage("1. Checkout Source Code") {  
    jenkinsfile_utils.checkoutSourceCode("PUSH")  
    def commitIdStdOut = sh(script: 'git rev-parse HEAD', returnStdout:  
true)  
    env.DEPLOY_GIT_COMMIT_ID = commitIdStdOut.trim()  
}
```

Stage Get ENV Productions: stage này thực hiện lấy thông tin cấu hình cho productions để kiểm tra quyền của user tác động đối với hệ thống.

```
stage("2. Get ENV Productions"){  
    sh '''  
        pwd  
        mkdir config-file  
        cd config-file  
        '''  
    dir('config-file'){  
        checkout changelog: true, poll: true, scm: [  
            $class : 'GitSCM',  
            branches : [[name: "master"]],  
            doGenerateSubmoduleConfigurations: false,  
            extensions : [[ $class: 'UserIdentity',  
email : 'hienptt22@viettel.com.vn', name:  
'hienptt22'],  
[ $class: 'CleanBeforeCheckout']],  
            submoduleCfg : [],  
            userRemoteConfigs : [[credentialsId: "63265de3-  
8396-40f9-803e-5cd0b694e519",  
url : "${env.config_productions}"]]  
        ]  
    }  
    sleep(5)  
    sh '''  
        ls -la
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**

**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 45/49

```
cp config-file/production-config.yml .
```

```
rm -rf config-file
```

```
cat production-config.yml
```

```
""
```

```
}
```

Stage Wait for maintainer accept or reject to deploy to production: stage này cấu hình thời gian và user có quyền tác động đến hệ thống. Xác định input là **Deploy** hay **Abort**.

```
def deployInput = "Deploy"
```

```
def deployer = ""
```

```
def config = readYaml file: "production-config.yml"
```

```
env.deployer_list = config['deployer_list']
```

```
env.ip_productions = config['ip_productions']
```

```
echo "Deploy List : ${env.deployer_list}"
```

```
echo "IP Productions : ${env.ip_productions}"
```

```
stage("3. Wait for maintainer accept or reject to deploy to production") {
```

```
try {
```

```
    deployer = env.deployer_list
```

```
    echo "project_maintainer_list: ${env.project_maintainer_list}"
```

```
    echo "deployer: ${deployer}"
```

```
    timeout(time: 24, unit: 'HOURS') {
```

```
        deployInput = input(
```

```
            submitter: "${deployer}",
```

```
            submitterParameter: 'submitter',
```

```
            message: 'Pause for wait maintainer selection', ok: "Execute",
```

```
parameters: [
```

```
    choice(choices: ['Deploy', 'Abort'], description: 'Deploy to  
production or abort deploy process?', name: 'DEPLOY_CHOICES')
```

```
    ]
```

```
}
```

```
    echo "submitter: ${deployInput.submitter}"
```

```
} catch (err) { // timeout reached or input false
```

```
    echo "Exception"
```

```
    def user = err.getCauses()[0].getUser()
```

```
    if ('SYSTEM' == user.toString()) { // SYSTEM means timeout.
```

```
        echo "Timeout is exceeded!"
```

```
    } else {
```



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 46/49

```
echo "Aborted by: [${user}]"  
}  
deployInput = "Abort"  
}  
echo "Input value: $deployInput"  
}
```

Stage Deploy to Productions: trong trường hợp user nhấn button **Deploy** là user **tác động hệ thống** thì sẽ thực hiện deploy tới môi trường productions.

```
if (deployInput.DEPLOY_CHOICES == "Deploy" &&  
deployer.contains(deployInput.submitter)) {
```

```
    stage('4. Deploy to Productions'){  
        echo "Tag: ${env.config_git_branch}"  
        untagVersionDeploy('auth',"${env.config_git_branch}")  
        untagVersionDeploy('ms-device',"${env.config_git_branch}")  
        untagVersionDeploy('ms-user-manage',"${env.config_git_branch}")  
        untagVersionDeploy('ms-app-manage',"${env.config_git_branch}")  
        untagVersionDeploy('service-discovery',"${env.config_git_branch}")  
        untagVersionDeploy('gateway',"${env.config_git_branch}")  
        jenkinsfile_CI.release2k8s('config_155_160','kafka-  
deployment_155_160.yml',"${env.config_git_branch}")  
    }
```

Stage Automations Testing after upcode: stage này thực hiện tự động kiểm tra lại luồng chính của ứng dụng để đảm bảo việc upcode tính năng mới không ảnh hưởng tới tính năng cũ. (Hoặc nếu tester/dev đã viết code test tính năng mới thì kết quả test sẽ kiểm tra toàn bộ các case test).

```
    stage("5. Automations Testing after upcode"){
```

```
        jenkinsfile_CI.autoTest("${env.ip_productions}","${env.tagsTestUpcode  
        }")  
    }
```

4.3. Hướng dẫn chi tiết từng bước cấu hình job jenkins và connect gitlab cho luồng CI/CD

Bước 1: Tạo một folder domain cho dự án chứa tất cả các job CI/CD của dự án đó

⇒ Tham khảo phần Cấu hình jenkins → [Bước 1](#)

Bước 2: Tạo một job CI/CD tương ứng

⇒ Tham khảo phần Cấu hình jenkins → [Bước 2](#)



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020
Ngày hết hiệu lực: 01/03/2021
Lần ban hành: 01
Trang/tổng số trang: 47/49

Bước 3: Cấu hình lựa chọn các Gitlab event mà job sẽ xử lý

⇒ Tham khảo phần Cấu hình jenkins → [Bước 3](#)

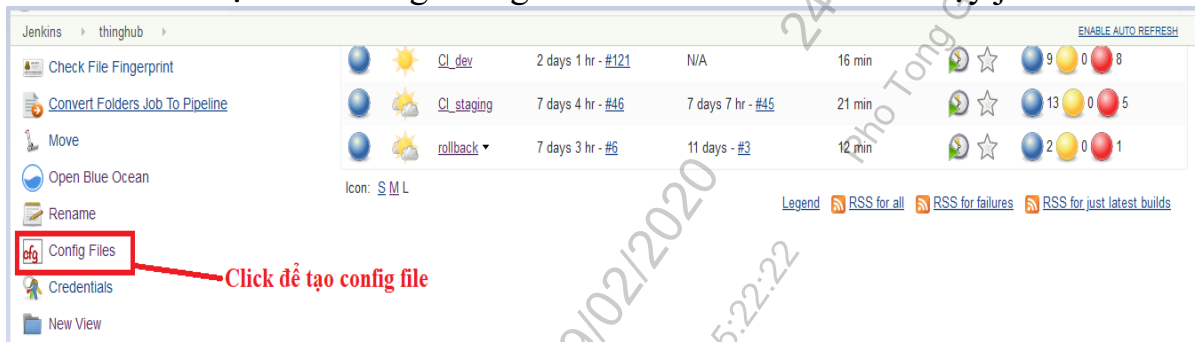
Bước 4: Cấu hình init script

⇒ Tham khảo phần cấu hình jenkins → [Bước 4](#)

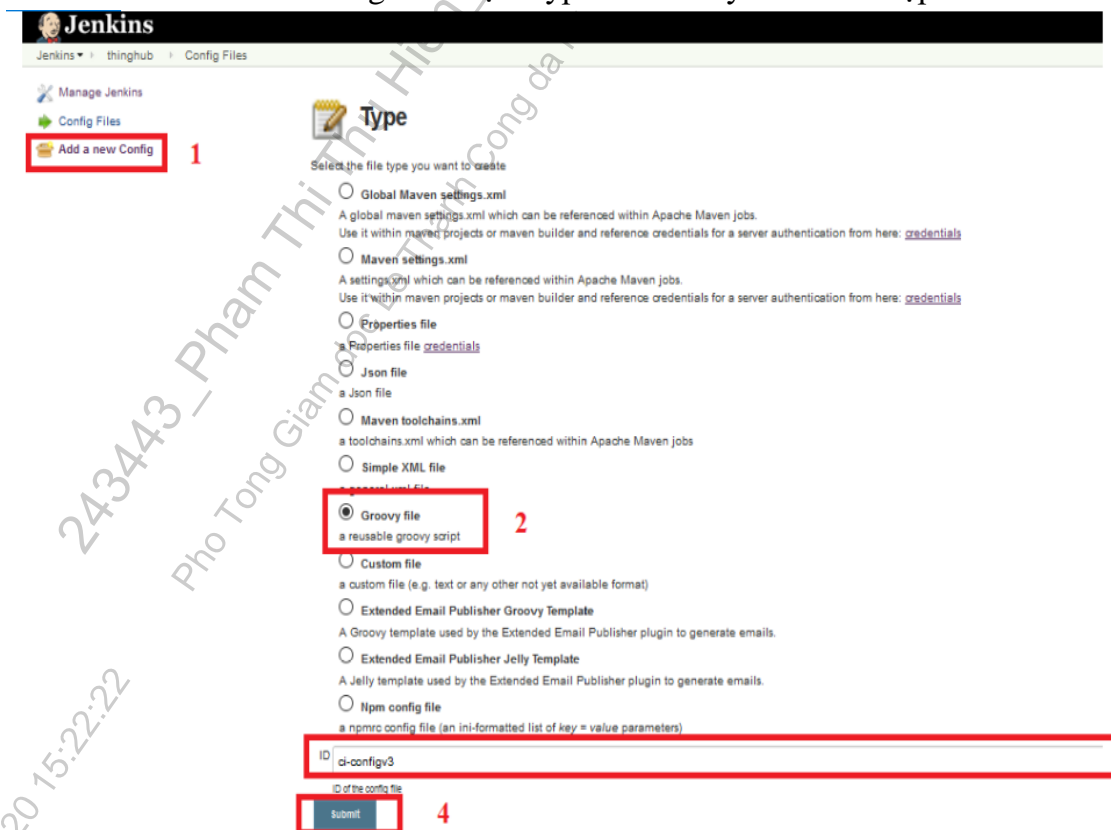
Bước 5: Cấu hình trigger trong gitlab

⇒ Tham khảo phần cấu hình gitlab → [Cấu hình gitlab](#)

Bước 6: Tạo file config chung cho cả folder Domain để chạy job



- **NOTE:** Config Files chỉ có khi tạo folder Job trên jenkins. Khi tạo job thông thường sẽ không có phần cấu hình Config File.
- Thực hiện cấu hình config file như sau: Tại folder job, Click Config Files → Add a new Config → Chọn Type là Groovy File → Nhập tên file



Cấu hình một số tham số cần thiết:



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 48/49



Edit Configuration File

The configuration

ID
ci-configv3

Name
ci-configv3

Comment

Content

```
1 env.NEXUS_REPO_URL = "10.60.156.26:8081"
2
3 //config sonar
4 env.MAXIMUM_ALLOWED_BUGS = 0
5 env.MAXIMUM_ALLOWED_VULNERABILITIES = 0
6 env.MAXIMUM_ALLOWED_CODE_SMELL = 0
7
8 env.bypass= 'true'
9 //env.bypass= 'false'
10
11 env.STAGING_BRANCH = 'master_cicd'
12
13
14 env.GITLAB_PROJECT_API_URL="http://10.60.156.11/api/v4/projects/459"
15
16 env.automations_test = "http://10.60.156.11/hiennt163/thinghub.git"
17
18 // Build prefix
19 env.PUSH_BUILD_PREFIX="JENKINS-PUSH"
20 env.MR_BUILD_PREFIX="JENKINS-MERGE"
21 env.ACCEPT_CLOSE_MR_BUILD_PREFIX="JENKINS-ACCEPT-CLOSE"
22
```

Submit

- Lưu ý: Cần cấu hình ID cho file để sử dụng ID trong hàm load-config-file trong script.

- Trong phần content cấu hình các biến môi trường sử dụng cho cả folder. Biến này có thể được sử dụng cho tất cả các job trong folder.

Bước 7: Tạo file pom.xml chung cho project

- Ở đây là project maven nên có thể tạo một file pom.xml để có thể xử lý các module trong project. Tham khảo tại mục [Project Layout](#)

- Đối với các loại project khác có thể customize cho phù hợp.

Bước 8: Copy các jenkinsfile cần thiết để chạy thử project. Bao gồm các file sau

```
jenkinsfile_bootstrap.groovy
jenkinsfile_CD.groovy
jenkinsfile_CI.groovy
jenkinsfile_utils.groovy
```

- Thực hiện lấy file script mẫu tại [đây](#).



**TỔNG CÔNG TY GIẢI PHÁP
DOANH NGHIỆP VIETTEL**
**HƯỚNG DẪN TRIỂN KHAI ÁP
DỤNG CI/CD PIPELINE TRONG
PHÁT TRIỂN PHẦN MỀM**

Mã hiệu: HD.VTS.GP.

Ngày có hiệu lực: 01/03/2020

Ngày hết hiệu lực: 01/03/2021

Lần ban hành: 01

Trang/tổng số trang: 49/49

NOTE: job chỉ chạy khi thực hiện push commit/merge request tới repository.

Không thực hiện test bằng cách nhấn button **build now trên jenkins →**
khi đó sẽ bị lỗi do jenkins không thể lấy được các biến môi trường của gitlab, ví
dụ : *lỗi không checkout được source code.*

4.4. Link tham khảo

- Link hướng dẫn public trên gitlab: <http://10.60.156.11/truongdxx/cicd-docs/blob/master/Module%20Config%20pipeline/README.md>